CS3025 Compiladores

Semana 5:
Acciones Semanticas / Semantic Actions
Arboles de Sintaxis Abstractos (ASTs)
12 Septiembre 2023

Igor Siveroni

Acciones Semánticas / Semantic Actions

- Un Compilador hace mas que reconocer si una cadena pertenece a un lenguaje o no – heos visto que esa es la labor del análisis sintáctico.

- Un Compilador debe hacer algo útil con la cadena!

- Las acciones semánticas de un analizador sintáctico (Parser) pueden realizar operaciones útiles con las partes de la cadena que son analizadas.

- ¿Cómo?
  En un parser de descenso recursivo, código relacionado a acciones semánticas es insertado al lado de las acciones que realizan el análisis.
  En una herramienta de genera analizadores automáticamente e.g. Bison, código puede ser agregado al lado de reglas de la gramática.

Acciones Semánticas / Semantic Actions

¿Esto que implica?

- En un parser de descenso recursivo, donde cada no-terminal esta relacionado a una función que retorna *void*, agregar acciones semánticas generalmente implica cambiar el tipo de retorno de estas funciones - a menos que la acción semántica implemente un side-effect como e.g. imprimir.

- A cada terminal y no terminal le asociamos un tipo para denotar valores semánticos, calculados por las acciones semánticas.
  Resultado: Cada función de análisis de (no)terminales ahora, además, calcula el “valor” de la palabra.

- Por ejemplo, si queremos agregar acciones semánticas para implementar una calculadora, los valores semánticos de los (no)terminals son enteros (`int`).

¡Pero esto ya lo vimos en los laboratorios!

Ejemplo: Expresiones Aritmeticas 1

La implementación del analizador sintáctico del lenguaje de expresiones aritméticas definido por la siguiente gramática:

```
Exp ::= Term ((+|-) Term)*
Term ::= Factor ((*|/) Factor)*
Factor ::= num | '(' Exp ')'
```

Posee la siguiente interface:

```
class Parser {
private:
  void parseExpression();
  void parseTerm();
  void parseFactor();
public:
  Parser(Scanner* scanner);
  void parse();
};
```

Si queremos agregarle acciones semánticas para evaluar la expresión aritmetica, el cambio a nivel de interface es:

```
class Parser {
private:
  int parseExpression();
  int parseTerm();
  int parseFactor();
public:
  Parser(Scanner* scanner);
  int parse();
};
```

Ejemplo: Expresiones Aritmeticas 2

La función que realiza el análisis sintáctico de `Factor` es:

```
void Parser::parseFactor() {
  if (match(Token::NUM)) {
    return;
  }
  if (match(Token::LPAREN)) {
    parseExpression();
    if (!match(Token::RPAREN)) {
      cout << "Error" << endl;
      exit(0);
    }
    return;
…
}
```

La nueva versión con acciones semánticas (para asignar valoes a las expresiones) es:

```
void Parser::parseFactor() {
 if (match(Token::NUM)) {
       return stoi(previous->lexema);
 }
 if (match(Token::LPAREN)) {
    int v = parseExpression();
    if (!match(Token::RPAREN)) {
      cout << "Error" << endl;
      exit(0);
    }
    return v;
  }
```

Ejemplo: Expresiones Aritmeticas 3

La función que realiza el análisis sintáctico de `Expression` es:

```
void
Parser::parseExpression() {
  parseTerm();
  while(match(Token::MINUS)||
        match(Token::PLUS)) {
    parseTerm();
  }
  return;
}
```

Y la versión con acciones semánticas es:

```
int Parser::parseExpression() {
  int accum, v;
  accum = parseTerm();
  while(match(Token::MINUS) ||
        match(Token::PLUS)) {
    Token::Type op = previous->type;
    v = parseTerm();
    if (op == Token::PLUS)
      accum += v;
    else
      accum -= v;
  }
  return accum;
}
```

Arboles de Sintaxis Abstractos (AST)

- Es posible escribir un compilador implementado con las acciones semánticas del parser. El principal problema con esta modalidad, además de ser difícil de leer y mantener, es que se restringe al compilador a ejecutar los análisis y demás operaciones exactamente en el orden en que los programas son analizados sintácticamente (”parseados”)
- Para mejorar la modularidad de un compilador, es mejor separar los aspectos de sintaxis (parsing) de los de semántica (análisis de tipos, generación de código, etc).
- Una manera de hacer esto es producir un árbol sintáctico – una estructura que el compilador podrá visitar varias veces, y en distinto orden.
- Ahora, el árbol sintactico debería incluir todos los no-terminales (nodo interno) usados durante la derivación de la cadena y tener como hojas a cada token generado por el parser.

¡Esto es demasiada información!

Arboles de Sintaxis Abstractos (AST)

- El árbol sintáctico, al que llamaremos *árbol sintáctico concreto*, posee, por ejemplo, signos de puntación que no agregan información: Estos símbolos son útiles para el análisis sintacttico pero una vez que el árbol es construido, toda la información necesaria esta en la estructura del árbol.
- La estructura del árbol puede depender mucha de la gramática!
  Las transformaciones a la gramática (eliminación de ambigüedades, asociatividad, recursión a la izquierda) introducidas para hacer posible el análisis sintáctico se vuelven innecesarias - es mas, obstruyen - para el análisis semántico que sigue a continuación. Estos detalles deben ser confinados a las primeras fases del compilador.

- La ***sintaxis abstracta*** provee una interface clara entre el análisis sintáctico y las siguientes fases del compilador.
- El árbol sintáctico abstracto (AST) contiene la estructura del programa fuente pero sin ninguna interpretación semántica.

Arboles de Sintaxis Abstractos (AST) – Sintaxis Abstracta

Por ejemplo, la sintaxis modificada que hacer posible el analisis sintactico de expresiones aritméticas unsado descenso recursivo es:

```
Exp ::= Term ((+|-) Term)*
        Term ::= Factor ((*|/) Factor)*
        Factor ::= num | '(' Exp ')'
```

Pero, una vez resueltos asociatividad y order de precedencia de operadores, solo es necesaria una gramática o sintaxis abstracta de la siguiente forma:

```
Exp ::= Exp (+|-|*|/) Exp |
        num |
        '(' Exp ')'
```

El compilador necesitara representar y manipular ASTs utilizando estructuras de datos.
Estrategia: Una clase abstracta por no-terminal y una subclase por cada producción.

Arboles de Sintaxis Abstractos (AST) – Implementacion

Siguiendo con el ejemplo de las expresiones aritméticas, los ASTs generados por la sintaxis

```
Exp ::= Exp (+|-|*|/) Exp | num |  '(' Exp ')'
```

Creamos una clase abstracta por el no-terminal Exp y una subclase por aca subproduccion:

```
enum BinaryOp { PLUS, MINUS, MULT,
DIV, EXP};

class Exp { …  virtual ~Exp() = 0;};

class BinaryExp : public Exp {
public:
  Exp *left, *right;
  BinaryOp op;
  BinaryExp(Exp* l, Exp* r, BinaryOp
op);
```

```
class NumberExp : public Exp {
public:
  int value;
  NumberExp(int v);
… };

class ParenthExp : public Exp {
public:
  Exp *e;
  ParenthExp(Exp *e);
… };
```

Arboles de Sintaxis Abstractos (AST) – Construccion

¿Cómo construimos los ASTs? ¡Con las acciones semanticas!

Los ASTs se construyen a traves de llamadas a constructores insertadas como acciones semantcias el el parser.

La nueva interface del Parser es:

```
class Parser {
private:
  Exp* parseExpression();
  Exp* parseTerm();
  Exp* parseFactor();
public:
  Parser(Scanner* scanner);
  Exp* parse();
};
```

Y la nueva implementación de parseFactor es:

```
Exp* Parser::parseFactor() {
  if (match(Token::NUM)) {
    strin s = previous->lexema;
    return new NumberExp(stoi(s));
  }
  if (match(Token::LPAREN)) {
   Exp* e = parseExpression();
   if (!match(Token::RPAREN)) {
      cout << ”Error" << endl;
      exit(0);
    }
   return new ParenthExp(e);
  }
… };
```

Arboles de Sintaxis Abstractos (AST) – Construccion

Del mismo modo actualizamos los otros metedos que realizan análisis sintactico y, desde ahora, construyen ATSs. Por ejemplo:

```
Exp* Parser::parseExpression() {
  Exp *e, *rhs;
  e = parseTerm();
  while(match(Token::MINUS) || match(Token::PLUS)) {
    Token::Type op = previous->type;
    BinaryOp binop = (op==Token::MINUS)? MINUS : PLUS;
    rhs = parseTerm();
    e = new BinaryExp(e, rhs, binop);
  }
  return e;
}
```

Arboles de Sintaxis Abstractos (AST) – Construccion

y,

```
Exp* Parser::parseTerm() {
  Exp *e, *rhs;
  e = parseFExp();
  while(match(Token::MULT) || match(Token::DIV)) {
    Token::Type op = previous->type;
    BinaryOp binop = (op==Token::MULT)? MULT : DIV;
    rhs = parseFExp();
    e = new BinaryExp(e, rhs, binop);
  }
  return e;
}
```

Nótese que seguimos analizando `Expression`s y `Term`s pero en ambos casos el valor retornado, el AST construido, es un `BinaryExp`.

Arboles de Sintaxis Abstractos (AST) – Recorrido / Visitas

Las siguientes fases del compilador se realizan via recorridos de los nodos internos y hojas del AST. Si, por ejemplo, queremos evaluar e imprimir y evaluar el árbol, debemos declarar las funciones correspondientes en la clase abstracta

```
class Exp {
public:
  virtual void print() = 0;
  virtual int eval() = 0;
  static string binopToString(BinaryOp op);
  virtual ~Exp() = 0;
};
```

E implementar los metodos en cada subclase. Las implementaciones para `NumberExp` y `ParenthExp` son :

```
int NumberExp::eval() {
  return value;
}

int ParenthExp::eval() {
  return e->eval();
}
```

Arboles de Sintaxis Abstractos (AST) – Recorrido / Visitas

Y para BinaryExp,

```
int BinaryExp::eval() {
  int v1 = left->eval();
  int v2 = right->eval();
  int result = 0;
  switch(this->op) {
  case PLUS: result = v1+v2; break;
  case MINUS: result = v1-v2; break;
  case MULT: result = v1 * v2; break;
  case DIV: result = v1 / v2; break;
  return result;
}
```

Arboles de Sintaxis Abstractos (AST) – Recorrido / Visitas

- Esto implica que, cada vez que tenemos que impementar una nuevo análisis o fase del compilador tenemos que:
  - Crear una nueva función en la clase abstracta
  - Implementar la función en todas las subclases

- Esto no es escalable, todas las funciones estarían asociadas explícitamente a las subclases del AST e implementadas en un solo archivo con todos los análisis (si implementamos las clases del AST en un solo archivo) o repartidas en varios archivos, uno por subclase.

- Lo que queremos es un archivo por análisis / fase del compilador.

Necesitamos implementar el Visitor Pattern
TBC

Laboratorio 5: Mas ASTs

Consideremos la siguiente gramática:

```
StmList ::= Stm (; Stm)*
Stm ::= id = Exp | print ‘(‘ Exp ‘)’
Exp ::= Term ((+|-) Term)*
Term ::= Fexp ((*|/) Fexp)*
Fexp ::= Factor  [‘**’ Fexp]
Factor ::= num | ‘(‘ Exp ‘)’ | id
```

extenderla para incluir expresiones condicionales de la forma

```
if-exp ´(´ Cexp , Exp, Exp ´)´
```

Donde, por ejemplo

```
x = 2;
y = if-exp(x > 4, 10, 20);
print(x+y)
```

Imprime 22.

Laboratorio 5

- La ultima versión del analyzador sintactico e interprete puede encontrarse en (semana 5) los archivos imp.hh, imp.cpp, imp_parser.hh, imp_parser.cpp e imp_test.cpp

- Que pasos son necesarios para implementar `if-exp` ?

- Cual es el problema si queremos extender la gramática para poder permitir sentencias como esta?

  y = x > 4

- Como podemos solucionarlo?